



DEVOPS E INTEGRAÇÃO CONTÍNUA

AULA 2



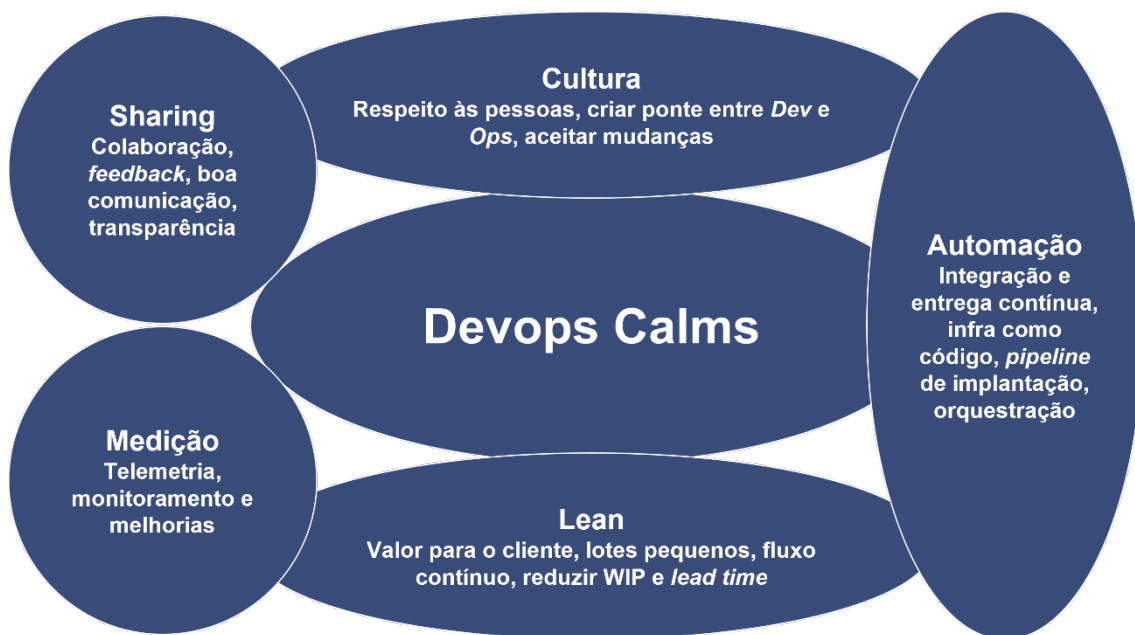
Prof. Mauricio Antonio Ferste

CONVERSA INICIAL

Logo nesta etapa, temos um ponto importante em nosso avanço de conhecimento, aprofundar o entendimento sobre versionamento e suas ferramentas na prática. Veremos pontos mais avançados na frente, mas a base será vista nesta unidade, e é essencial seu entendimento.

Anteriormente, vimos muito da teoria de DevOps, como dado por

Figura 1 – O conceito de CALMS DevOps



Fonte: Muniz, 2019.

CALMS é uma estrutura que avalia a capacidade de uma empresa adotar processos de DevOps, bem como uma maneira de medir o sucesso durante uma transformação de DevOps. A sigla foi cunhada por Jez Humble, coautor de *The DevOps Handbook*, e significa **Cultura, Automação, Lean, Medição e Compartilhamento**.

Agora teremos de começar a pensar no funcionamento e adoção de todos esses processos para fazer versionamento. Quando iniciarmos o estudo de Git, GitHub, GitLab e outros, estaremos abraçando a prática do processo, ou seja, não é só versionar arquivo, é entender de fluxo de criação, recuperação e várias ferramentas.



TEMA 1 – INTRODUÇÃO AO CONTROLE DE VERSÃO

Chegamos a um ponto em que é prático para todos usarem sistemas de controle de versão em seu trabalho. O *Subversion* é um sistema disponível gratuitamente que suporta facilmente formatos binários e remove muitas das limitações do CVS (Fowler, 2023).

O espaço em disco é barato o suficiente para que você possa colocar todo o diretório de trabalho das pessoas sob controle de versão. O controle de versão é uma prática essencial no desenvolvimento de *software* que visa gerenciar as alterações realizadas em um projeto ao longo do tempo. Ele proporciona uma maneira organizada e controlada de rastrear as modificações no código-fonte, facilitando a colaboração entre desenvolvedores e a manutenção do histórico de alterações.

Essa prática é fundamental para evitar conflitos entre diferentes contribuições ao mesmo projeto, permitindo que equipes trabalhem simultaneamente em partes distintas do código (Git-Fast-Version-Control, 2023). Além disso, o controle de versão possibilita reverter para versões anteriores do código em caso de problemas, garantindo a estabilidade do projeto.

Os sistemas de controle de versão, como Git e SVN, são amplamente utilizados para implementar essas funcionalidades. Eles proporcionam uma estrutura para o armazenamento, rastreamento e colaboração eficiente, essenciais para o desenvolvimento de *software* moderno.

Saiba mais

Para ilustrar podemos ver, por exemplo, no YouTube, bons vídeos complementares sobre controle de versão, como a seguir. Disponível em: <<https://www.youtube.com/watch?v=FlulsydqsJM>>. Acesso em: 15 abr. 2024.

1.1 Definições

Controle de versão é uma ferramenta que registra todas as alterações feitas em um arquivo ou conjunto de arquivos ao longo do tempo. Isso permite que você recupere versões anteriores do arquivo, compare alterações entre versões e visualize quem fez as alterações (Chacon, 2017).

O controle de versão é uma prática essencial para qualquer desenvolvedor de *software*. Ele permite que você rastreie as alterações feitas



em seus arquivos ao longo do tempo, reverta para versões anteriores se necessário e colabore com outras pessoas.

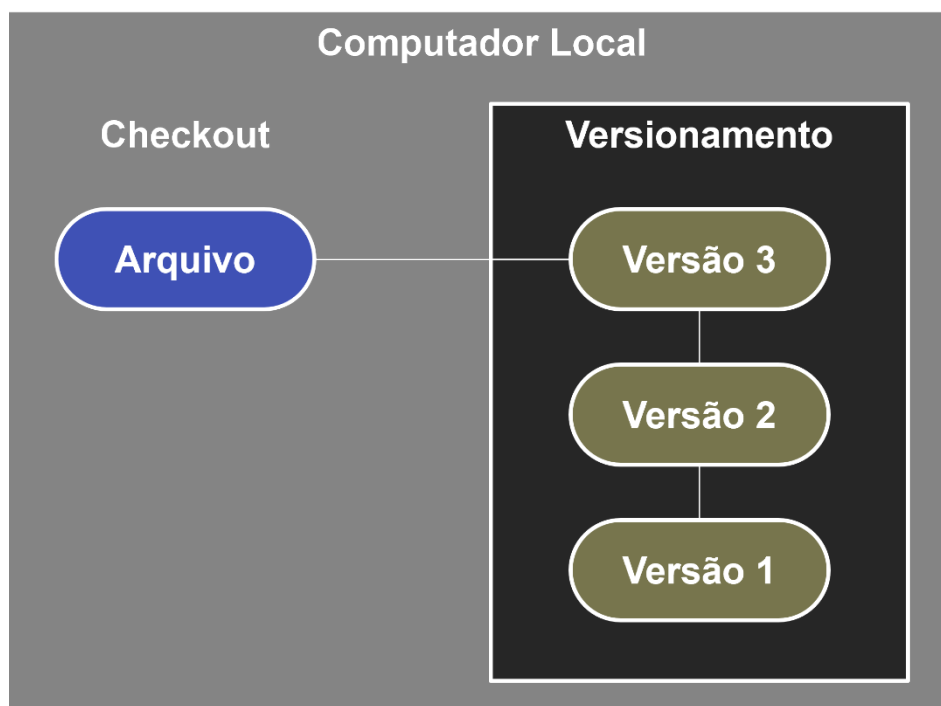
Existem muitos métodos diferentes de controle de versão. Um método comum é copiar os arquivos para um diretório separado. Essa abordagem é simples, mas tem algumas desvantagens. É fácil esquecer em qual diretório você está resultando na perda acidental do arquivo errado ou na cópia de arquivos indesejados.

Para evitar esses problemas, os programadores há muito tempo usam Sistemas de Controle de Versão Locais (VCSs locais). Os VCSs locais mantêm um banco de dados de todas as alterações feitas nos arquivos sob controle de revisão. Isso permite que você veja facilmente o que foi alterado, quando foi alterado e quem fez as alterações.

1.2 Benefícios do Controle de Versão

Imagine que você está trabalhando em um projeto de *design* gráfico e cria uma nova imagem. Você está satisfeito com o resultado, mas decide fazer algumas alterações. Depois de fazer as alterações, você percebe que não gosta do resultado. Com controle de versão, você pode facilmente reverter para a versão anterior da imagem, como na Figura 2.

Figura 2 – Versionamento em um computador local



Fonte: elaborada por Ferste, 2024, com base em Git-Fast-Version-Control, 2023.



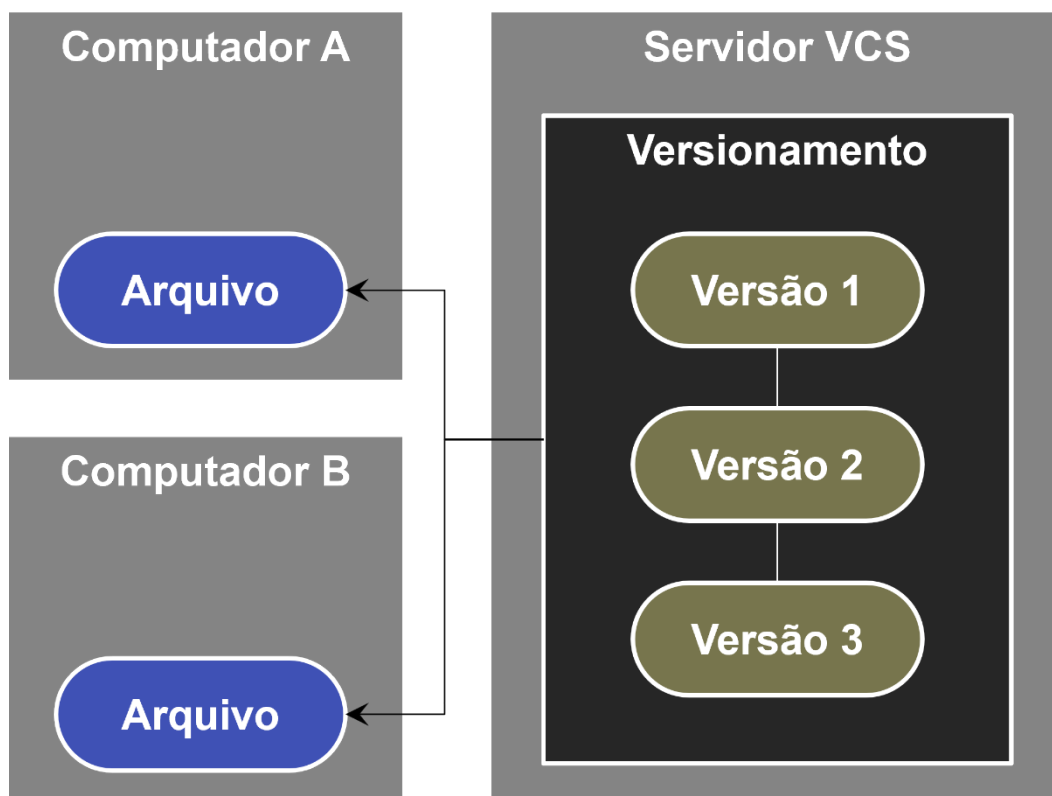
A seguinte questão crucial que muitas pessoas enfrentam é a necessidade de colaborar com desenvolvedores que estão em outros sistemas. Para resolver esse problema, foram desenvolvidos Sistemas Centralizados de Controle de Versão (CVCSs).

Os Sistemas Centralizados de Controle de Versão (CVCSs) são uma solução para a questão crucial de colaborar com desenvolvedores que estão em outros sistemas. Esses sistemas armazenam todos os arquivos de controle de versão em um único servidor, que é acessado por vários clientes.

CVCSs

- Facilidade de colaboração: os desenvolvedores podem trabalhar nos mesmos arquivos ao mesmo tempo, sem a necessidade de sincronizar manualmente seus arquivos.
- Controle de acesso: os administradores podem controlar quem tem acesso aos arquivos de controle de versão.
- *Backups*: os arquivos de controle de versão são armazenados centralmente, o que facilita o *backup* e a recuperação.

Figura 3 – Versionamento em um servidor compartilhado



Fonte: elaborada por Ferste, 2024, com base em Git-Fast-Version-Control, 2023.



Exemplos de CVCSs:

- CVS
- *Subversion*
- *Perforce*

Os CVCSs são a norma padrão para o controle de versão há muitos anos. No entanto, eles também apresentam algumas desvantagens, como dependência de um servidor: os CVCSs dependem de um servidor central, o que pode ser um ponto de falha.

Desempenho: os CVCSs podem ter um desempenho inferior aos sistemas distribuídos.

DVCSs

Os Sistemas Distribuídos de Controle de Versão (DVCS) oferecem uma solução para a dependência de um servidor central dos Sistemas Centralizados de Controle de Versão (CVCSs). Em um DVCS, cada cliente possui uma cópia local do repositório completo. Isso significa que, se um servidor falhar, os clientes ainda poderão acessar os arquivos de controle de versão.

Os DVCS oferecem várias vantagens em relação aos CVCS, como resistência a falhas, pois os DVCS não dependem de um servidor central, o que os torna mais resistentes a falhas. Eles têm mais desempenho, pois podem oferecer melhor desempenho do que os CVCSs, pois as operações não precisam ser transmitidas por uma rede. Também tem mais autonomia, os clientes de DVCS podem trabalhar *offline*, sem a necessidade de estar conectados a um servidor central.

Exemplos de DVCSs:

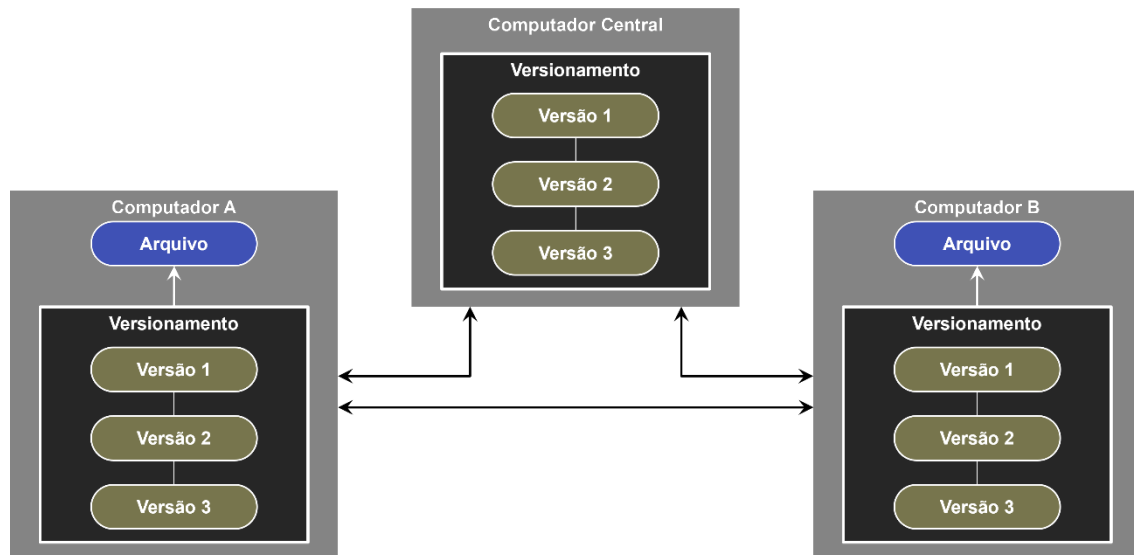
- Git
- Mercurial
- Bazaar
- Darcs

Os DVCS são amplamente utilizados em projetos de *software* de código aberto, em que é importante garantir a resistência a falhas e a autonomia. Eles também são usados em projetos corporativos, em que a disponibilidade é uma prioridade.



Como conclusão, os DVCS são uma opção viável para projetos que precisam de um sistema de controle de versão resistente a falhas e com bom desempenho.

Figura 4 – Versionamento em um DVCS



Fonte: elaborada por Ferste, 2024, com base em Git-Fast-Version-Control, 2023.

TEMA 2 – GIT E GITHUB: O BÁSICO

2.1 Entendendo Melhor GIT

O Git é o sistema de controle de versão mais popular do mundo. Ele é um projeto de código aberto maduro e ativamente mantido, desenvolvido por Linus Torvalds, o criador do kernel do Linux. O Git é usado em uma ampla variedade de projetos de *software*, tanto comerciais quanto de código aberto. O Git é um sistema de controle de versão distribuído (DVCS).

Isso significa que cada desenvolvedor tem uma cópia local do repositório completo. Ao contrário dos sistemas de controle de versão centralizados (CVCSs), que armazenam o histórico completo do *software* em um único servidor, o Git permite que os desenvolvedores trabalhem *offline* e colaborem com mais facilidade.

O Git, um sistema de controle de versão, foi concebido por Linus Torvalds em colaboração com a comunidade Linux. Seu objetivo era oferecer uma alternativa ao *Bitkeeper* e superar as limitações presentes em sistemas concorrentes. A necessidade de desenvolver um novo sistema de versionamento de *software* surgiu em 2005, quando a equipe do *BitKeeper* retirou sua



ferramenta paga da comunidade Linux, alegando violação de licença. Diante desse cenário, Linus e a comunidade Linux sentiram a necessidade de criar seu próprio sistema para gerenciar as versões do kernel do Linux.

Figura 5 – Logo do Git idealizada por Linus Torvalds



Crédito: Postmodern Studio/Shutterstock.

2.2 Conhecendo GitHub

O GitHub é uma plataforma de hospedagem de código-fonte que fornece ferramentas para desenvolvedores colaborarem em projetos de *software*. Basicamente, existem hoje vários sítios baseados em Git que podem ser utilizados, de longe o mais famoso e conhecido é o GitHub, ou seja, o GitHub hospeda repositórios Git, mas não é o único, que são o local em que os desenvolvedores armazenam seu código-fonte.

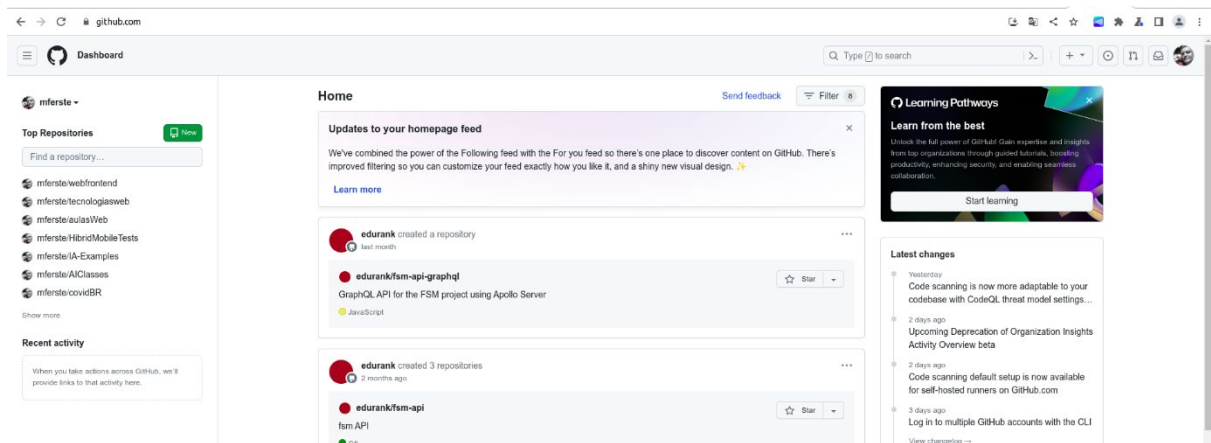
Existem vários repositórios Git pelo mundo, até mesmo a Azure da Microsoft ou AWS tem seus próprios repositórios, várias empresas mantêm seus repositórios próprios, portanto, não se deve pensar no GitHub como única opção, embora seja de longe a pública mais utilizada no mundo hoje e lembrando que existem também opções pagas para o GitHub com serviços especializados.

Por exemplo, considerando Azure temos o Azure Repos <<https://azure.microsoft.com/pt-br/products/devops/repos>>, que é um exemplo de uso pago de Git. (Azure Repos, 2023). O mesmo tem a Amazon, com o CodeCommit <<https://aws.amazon.com/pt/codecommit/>> (Amazon Code Commit, 2023).

Agora vamos de fato ao GitHub...



Figura 6 – Tela de um projeto no GitHub



Crédito: Mauricio Antonio Ferste.

O GitHub desempenha um papel crucial no ecossistema de desenvolvimento de *software*, proporcionando uma plataforma centralizada e uma variedade de ferramentas de colaboração, como problemas, solicitações *pull* e revisão de código. Essas ferramentas ajudam os desenvolvedores a se comunicar, compartilhar *feedback* e revisar mudanças.


2.3. Iniciando na prática com GitHub

Neste tópico, vamos passar os conhecimentos iniciais relacionados ao fato de começar o trabalho. Como iniciar um projeto como seu lugar? Como realizar algumas operações básicas que são necessárias para montar o seu acesso? Que pontos específicos temos de ter em mente e temos que ter cuidado?


Primeiramente, ao acessar o GitHub, temos que fazer a autenticação. Autenticação é valiosa, representa o valor que damos ao nosso código. Então temos que pensar que hoje existem várias formas de autenticação com fator duplo? Como no caso da Figura 7 a seguir. Podemos ver a necessidade desta autenticação.





Figura 7 – Autenticação em dois ou mais fatores



Two-factor authentication



Authentication code 



Open your two-factor authenticator (TOTP) app or browser extension to view your authentication code.

Having problems?

- [Use a recovery code or begin 2FA account recovery](#)

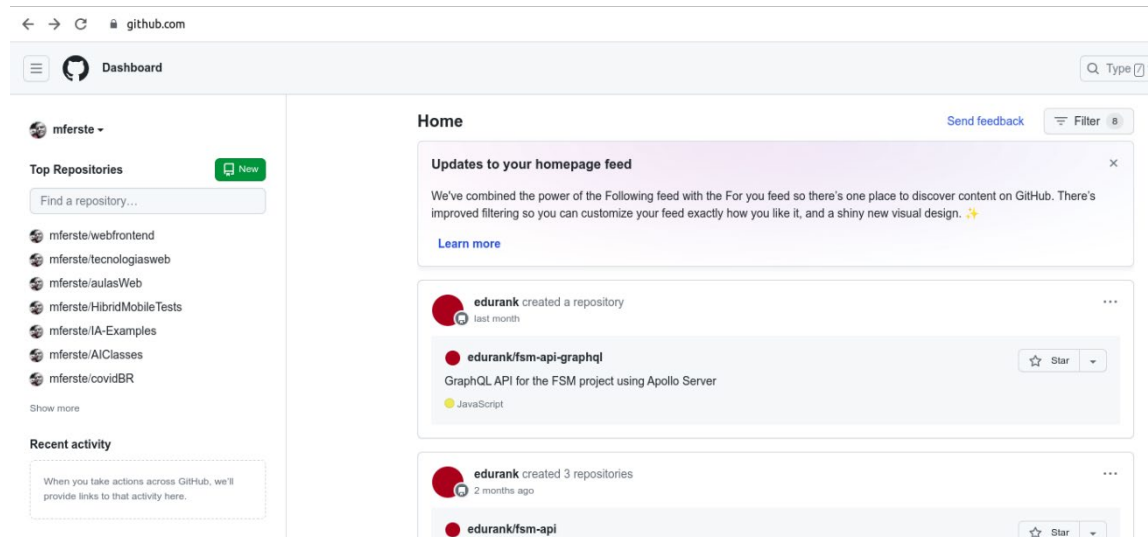
Crédito: Mauricio Antonio Ferste.

Esse ponto pode exigir a instalação de programas muitas vezes no seu celular. Para a validação, um muito utilizado é o *authenticator*, então é interessante pensar que você vai precisar utilizar o seu *e-mail*, seu telefone e instalação de programas como *authenticator* que hoje são muito necessários no fato de você tratar o acesso no meio que é público e garantir a segurança do seu código.

Uma vez logado, podemos passar para a criação do projeto. O projeto é criado clicando em *new*, no seu menu da sua barra de navegação. Como na Figura 8 a seguir.



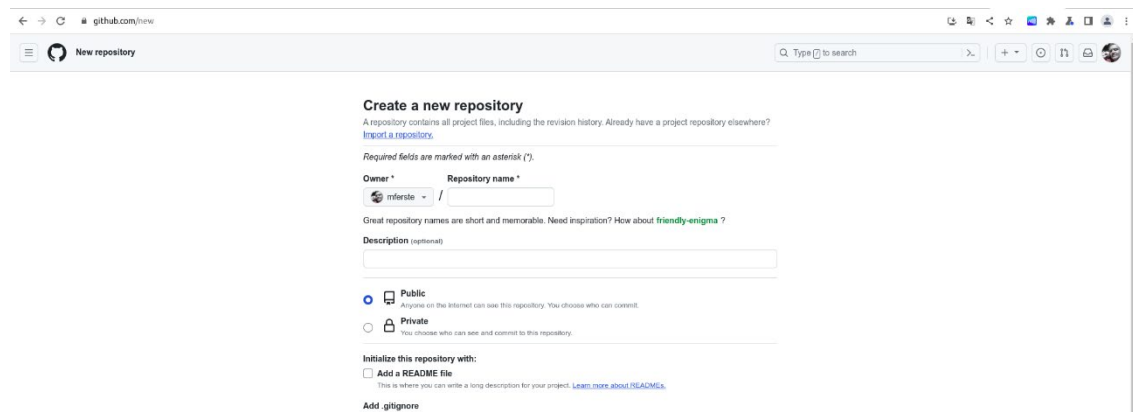
Figura 8 – Criando Projetos



Crédito: Mauricio Antonio Ferste.

Após ter pressionado ou teclado em *New*, podemos ver que na Figura 9 o resultado da criação do projeto tem algumas configurações básicas que podem e devem ser ajustadas, quanto ao nome do projeto, o fato de o mesmo ser público ou privado.

Figura 9 – Processo de criação de um projeto



Crédito: Mauricio Antonio Ferste.

Detalhando conforme a Figura 9, temos:

- **Nome do Projeto:** uma breve descrição do seu projeto vai aqui.
- **Introdução:** explique o propósito do seu projeto. Qual problema ele resolve? Por que é valioso?
- **Recursos:** liste os principais recursos do seu projeto.



- **Instalação:** forneça instruções passo a passo sobre como instalar e configurar seu projeto localmente.

Exemplo de comando de instalação ou trecho de código

- **Uso:** descreva como usar seu projeto. Inclua exemplos, trechos de código ou capturas de tela, se aplicável.
- **Contribuição:** explique como outras pessoas podem contribuir para o seu projeto. Inclua diretrizes para relatórios de *bugs*, solicitações de recursos e contribuições de código.
- **Licença:** indique a licença sob a qual seu projeto é lançado.

Existe também o arquivo **.gitignore**, que detalharemos melhor mais à frente. O arquivo **.gitignore** é usado no sistema de controle de versão Git para especificar quais arquivos e pastas devem ser ignorados e não incluídos no repositório Git. Isso é útil para evitar que arquivos temporários, arquivos gerados automaticamente, arquivos de compilação e outros artefatos desnecessários sejam rastreados pelo Git. Ao listar esses padrões no arquivo **.gitignore**, você instrui o Git a ignorar automaticamente os arquivos correspondentes durante operações como **git add** e **git commit**.

TEMA 3 – DOMINANDO *BRANCHING*, *MERGING* E *PULL REQUEST*

3.1 Principais comandos do Git

Para utilizar o Git, os desenvolvedores empregam comandos específicos para copiar, criar, alterar e combinar código. Essas ações podem ser realizadas diretamente na linha de comando ou por meio de aplicativos como o GitHub Desktop. Alguns comandos comuns para usar o Git incluem:

Inicializar um novo repositório Git e começar a rastrear um diretório existente usando **git init**. Esse comando adiciona uma subpasta oculta ao diretório existente que abriga a estrutura de dados interna necessária para o controle de versão, criando uma cópia local de um projeto já existente remotamente com **git clone**. O clone inclui todos os arquivos, histórico e ramificações do projeto.

Encenar uma mudança com **git add**: o Git rastreia as alterações na base de código de um desenvolvedor, mas é necessário preparar e tirar um instantâneo das alterações para incluí-las no histórico do projeto. Esse comando

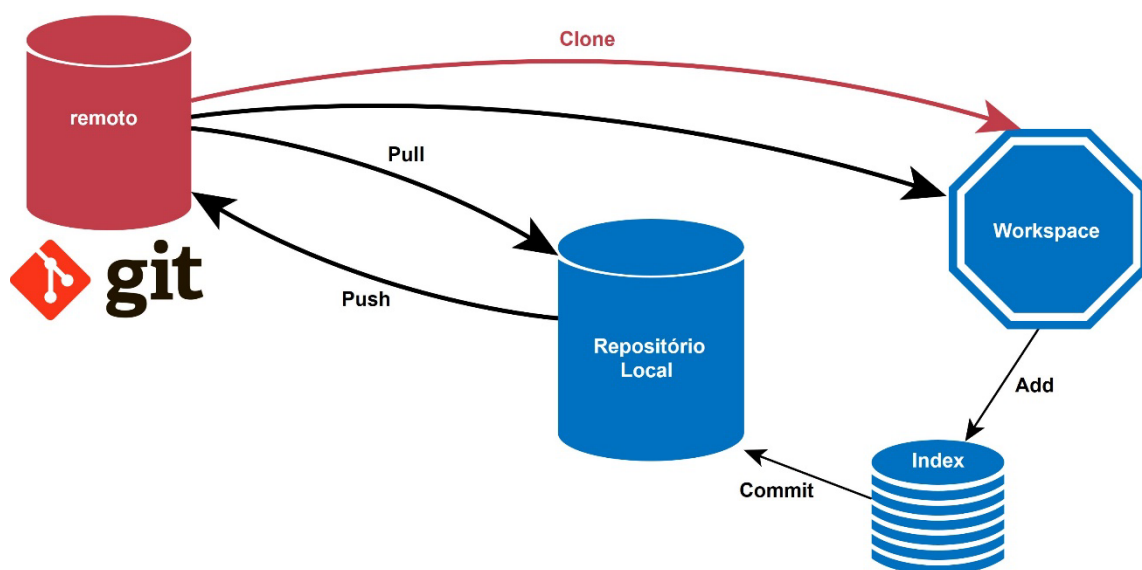


executa a preparação, a primeira parte desse processo de duas etapas. Quaisquer alterações preparadas se tornarão parte do próximo *snapshot* e parte do histórico do projeto. Preparar e confirmar separadamente dá aos desenvolvedores controle total sobre o histórico de seu projeto sem alterar a forma como codificam e trabalham.

Salvar o instantâneo no histórico do projeto e concluir o processo de controle de alterações com **git commit**. Resumidamente, um commit funciona como tirar uma foto. Qualquer coisa que tenha sido testada com **git add** se tornará parte do instantâneo com **git commit**.

Verificar o *status* das alterações, como não rastreadas, modificadas ou preparadas, com **git status**. Mostrar as filiais que estão sendo trabalhadas localmente com **git branch**.

Figura 10 – Funcionamento de principais comandos e operações do GIT



Crédito: Arte/UT.

Fundir linhas de desenvolvimento usando **git merge**. Esse comando é geralmente usado para combinar alterações feitas em duas ramificações distintas, por exemplo, quando um desenvolvedor deseja unir alterações de uma ramificação de recurso na ramificação principal para implantação.

Atualizar a linha local de desenvolvimento com atualizações de sua contraparte remota com **git pull**. Os desenvolvedores utilizam esse comando se um colega de equipe fez *commits* em uma ramificação remota e gostariam de refletir essas alterações em seu ambiente local. Atualizar o repositório remoto com quaisquer *commits* feitos localmente em uma ramificação com **git push**.



3.2 Comandos de fluxo de trabalho, *branch*, *merge*, *pull*

Dominar *branching*, *merging* e *pull requests* é essencial para um fluxo de trabalho eficiente no Git. Vamos explorar esses conceitos:

- **Branching (ramificação):** *branching* permite que você crie linhas separadas de desenvolvimento, chamadas de "ramificações", para trabalhar em funcionalidades ou correções sem afetar a linha principal (geralmente chamada de "*master*" ou "*main*"). Ramificar isola o trabalho em progresso, permitindo que diferentes equipes ou desenvolvedores trabalhem simultaneamente em recursos distintos sem interferências.
- **Merging (mesclagem):** *merging* é o processo de combinar as alterações de uma ramificação em outra. Isso geralmente ocorre quando uma funcionalidade é concluída e precisa ser incorporada de volta à linha principal do desenvolvimento. Mesclar garante que as alterações de diferentes ramificações sejam incorporadas corretamente, evitando conflitos e mantendo a integridade do código.
- **Pull Requests (solicitações de Pull):** uma *pull request* é uma solicitação para mesclar as alterações de uma ramificação em outra, geralmente de uma *branch* de recurso para a *branch* principal. As *pull requests* fornecem um meio estruturado para revisar códigos antes da mesclagem. Permitem discussões, revisões de código e testes antes que as alterações sejam incorporadas à linha principal.

3.2.1 Fluxo de trabalho típico

Para criar um "ambiente", normalmente temos de seguir uma ordem que tende a utilizar os comandos abaixo:

- **Criação de Branch:** inicie uma nova ramificação para trabalhar em uma funcionalidade ou correção.
- **Commits:** faça *commits* regulares na sua ramificação local para registrar o progresso.
- **Push:** envie a ramificação para o repositório remoto para colaboração e *backup*.
- **Pull Requests:** abra uma *pull request* para iniciar a revisão do código e a discussão.



- **Code Review:** outros membros da equipe revisam o código e fornecem *feedback*.
- **Merge:** após a aprovação, a ramificação é mesclada na *branch* principal.

Dominar esses conceitos não apenas facilita um fluxo de trabalho suave, mas também promove uma colaboração eficiente em equipes de desenvolvimento.

3.3 Vamos criar um repositório

Seguem registros de como criar um repositório.

```
# criando um novo diretório e inicializando
git init meu-repositorio

# acessando...via texto
cd my-repo

# criando o primeiro arquivo, que normalmente é o README.md
touch README.md

# git não está ciente do arquivo, preparando-o
git add README.md

# registrando um momento sobre o ponto atual
git commit -m "adicionando o READ.me"

# adicionando a origem do repositório no github
git remote add origin https://github.com/SEU-USUARIO/SEU-REPOSITORIO.git

# enviando alterações para o GitHub
git push --set-upstream origin main
```

3.4 Agora acessando um repositório já existente

Seguem registros acessando um repositório.

```
# Trocar 'owner/repositorio' com o proprietário do repositório
para clonar
git clone https://github.com/owner/repo.git

# no caso de nosso curso temos um repositório chamado
```



```
# https://github.com/N-CPUinter/DevOps.git

# entrando no repositório clonado
cd repo

# criando uma nova versão/ramo para trabalhar
git branch meu-branch

# mudar para esse ramo (linha de desenvolvimento)
git checkout meu-branch

#
# fazendo as mudanças necessárias quaisquer que sejam
#

# adicionando os arquivos modificados
git add file1.md file2.md

# criando um registro
git commit -m "meu-branch"

# enviando para o GitHub
git push --set-upstream origin meu-branch
```

TEMA 4 – FERRAMENTAS DE APOIO PARA VERSIONAMENTO

4.1 Exemplos de Comandos para versionamento em linha de comando

O uso de ferramentas de versionamento é essencial para o controle eficiente de código-fonte em projetos de desenvolvimento de *software*. Aqui estão os passos básicos para utilizar ferramentas de versionamento, como Git, que é uma das mais populares:

Instalação da Ferramenta: comece instalando a ferramenta de versionamento desejada, como o Git. Você pode baixar e instalar a versão adequada para o seu sistema operacional no *site* oficial.

- **Configuração Inicial:** após a instalação, configure seu nome de usuário e endereço de *e-mail* usando os comandos:

```
git config --global user.name "Seu Nome"
git config --global user.email "seu@email.com"
```

- **Criação de um Repositório Local:** vá até o diretório do seu projeto no terminal e inicie um repositório Git local com o comando:

```
git init
```




- **Adição de Arquivos ao Repositório:** adicione os arquivos do seu projeto ao controle de versão com o comando:

```
git add .
```

- **Confirmação de alterações:** realize uma confirmação (*commit*) para salvar as alterações adicionadas. Adicione uma mensagem descritiva ao comando:

```
git commit -m "Mensagem descritiva das alterações"
```

- **Trabalhando com Ramificações:** se necessário, crie e alterne entre ramificações para desenvolver recursos separadamente, utilizando comandos como *git branch* e *git checkout*.
- **Sincronização com Repositório Remoto:** se você estiver colaborando com outros desenvolvedores ou usando serviços de hospedagem remota (GitHub, GitLab, Bitbucket), configure e adicione um repositório remoto:

```
git remote add origin URL-do-Repositório-Remoto  
git push -u origin master
```

- **Atualizações e Mesclagem:** mantenha seu repositório local atualizado com alterações remotas usando *git pull*. Realize a mesclagem (*merge*) de ramificações quando necessário.
- **Resolução de Conflitos:** se ocorrerem conflitos durante a mesclagem, resolva-os manualmente, adicione as alterações e faça um novo *commit*.
- **Histórico e Visualização:** utilize comandos como *git log* para visualizar o histórico de commits e *git diff* para ver as diferenças entre versões.

Esses passos básicos ajudarão você a começar a utilizar ferramentas de versionamento. Conforme você se torna mais familiarizado, pode explorar funcionalidades avançadas e integrar-se a fluxos de trabalho mais complexos, como integração contínua e entrega contínua (CI/CD).

Ferramentas de apoio ao versionamento desempenham um papel crucial no desenvolvimento de *software*, oferecendo recursos para gerenciar eficientemente o controle de versão e facilitar a colaboração entre membros da equipe. Essas ferramentas podem ser categorizadas em diversas áreas, como plataformas de hospedagem de código, interfaces gráficas de usuário (GUI),



ambientes de desenvolvimento integrado (IDE), ferramentas de linha de comando, sistemas de integração contínua, ferramentas de revisão de código e ferramentas de rastreamento de problemas.

Importante: O Git Credential Manager (GCM) é uma ferramenta que ajuda a gerenciar e armazenar credenciais de forma segura para o Git. Ele é especialmente útil em sistemas operacionais Windows, em que pode ser mais desafiador configurar e armazenar credenciais de forma segura.

No contexto do Git, o SSH é amplamente utilizado para autenticar e se conectar de forma segura a repositórios Git hospedados em servidores remotos. Essa abordagem é particularmente valiosa para acessar repositórios privados, garantindo que apenas usuários autorizados possam interagir com o código fonte. Ao utilizar o Git com SSH, é comum configurar chaves SSH para autenticação nos servidores remotos. Essas chaves, normalmente compostas de uma chave pública e uma chave privada, garantem uma forma segura de autenticação, sem a necessidade de fornecer constantemente credenciais de login.

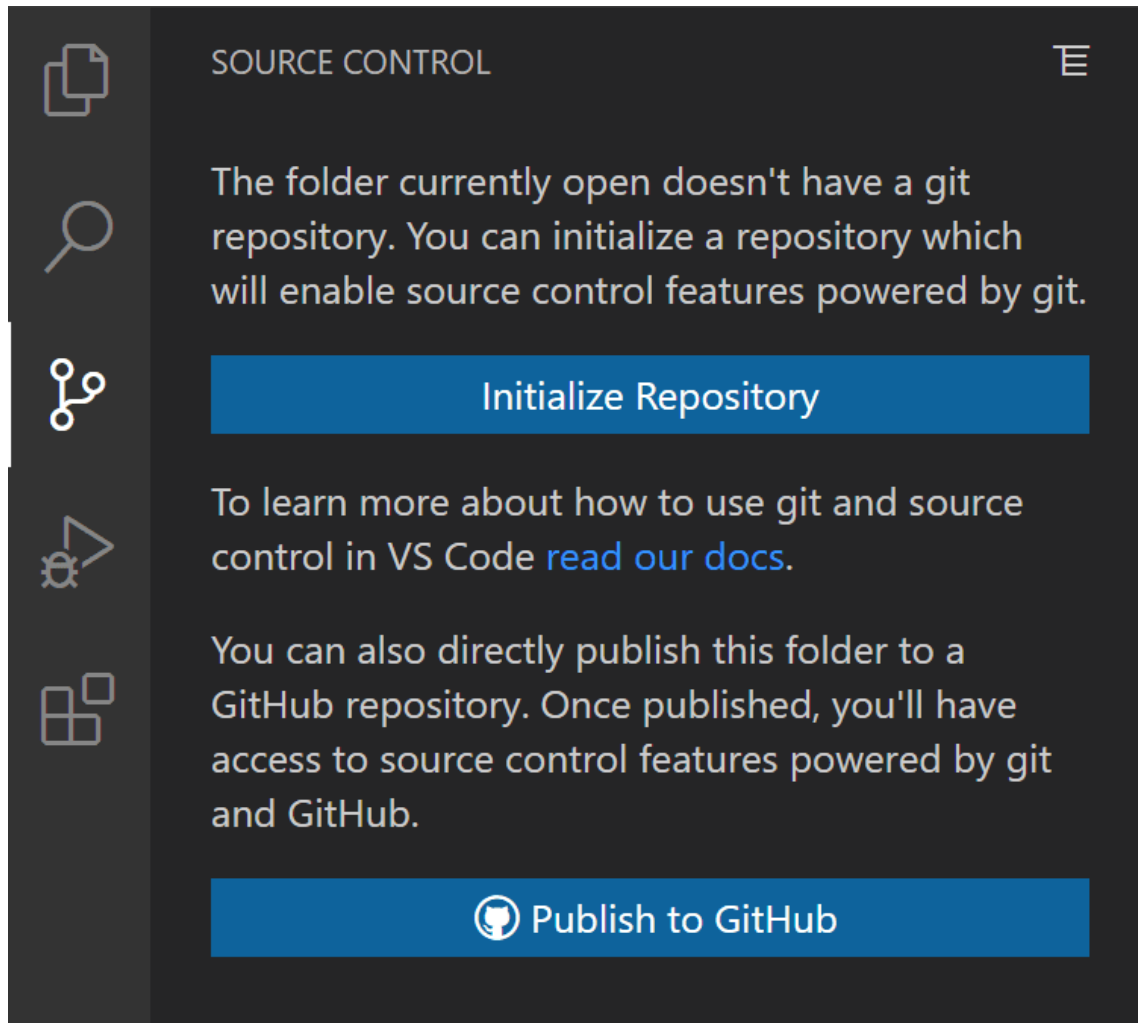
Por outro lado, o HTTP (*Hypertext Transfer Protocol*) é um protocolo de comunicação usado para transferir dados pela internet. No contexto do Git, o HTTP oferece uma alternativa ao SSH para acessar repositórios remotos. Isso é especialmente útil ao interagir com repositórios hospedados em serviços populares como GitHub, GitLab ou Bitbucket, que suportam acesso via HTTP.

Ao usar o Git com HTTP, pode ser necessário fornecer suas credenciais (nome de usuário e senha) sempre que interagir com o repositório. Em alguns casos, pode ser necessário configurar autenticação de dois fatores ou gerar um *token* de acesso pessoal para garantir a segurança da conexão. Essa abordagem é conveniente para muitos usuários, mas pode exigir a digitação frequente de credenciais ao interagir com os repositórios.

Em resumo, tanto o SSH quanto o HTTP são opções viáveis para acessar repositórios Git remotos, cada um com suas próprias vantagens e considerações de segurança. A escolha entre eles dependerá das necessidades específicas do projeto e das políticas de segurança da organização.



Figura 11 – O uso de ferramentas para o trabalho com GIT



Fonte: <<https://code.visualstudio.com/>>. Acesso em: 15 abr. 2024.

Plataformas de hospedagem de código, como GitHub e GitLab, proporcionam repositórios na nuvem com recursos integrados de colaboração. Interfaces gráficas de usuário, como SourceTree e GitKraken, simplificam as interações com o controle de versão por meio de uma interface visual. Ambientes de desenvolvimento integrado, como *Visual Studio Code*, incorporam funcionalidades de versionamento diretamente no fluxo de trabalho de desenvolvimento.

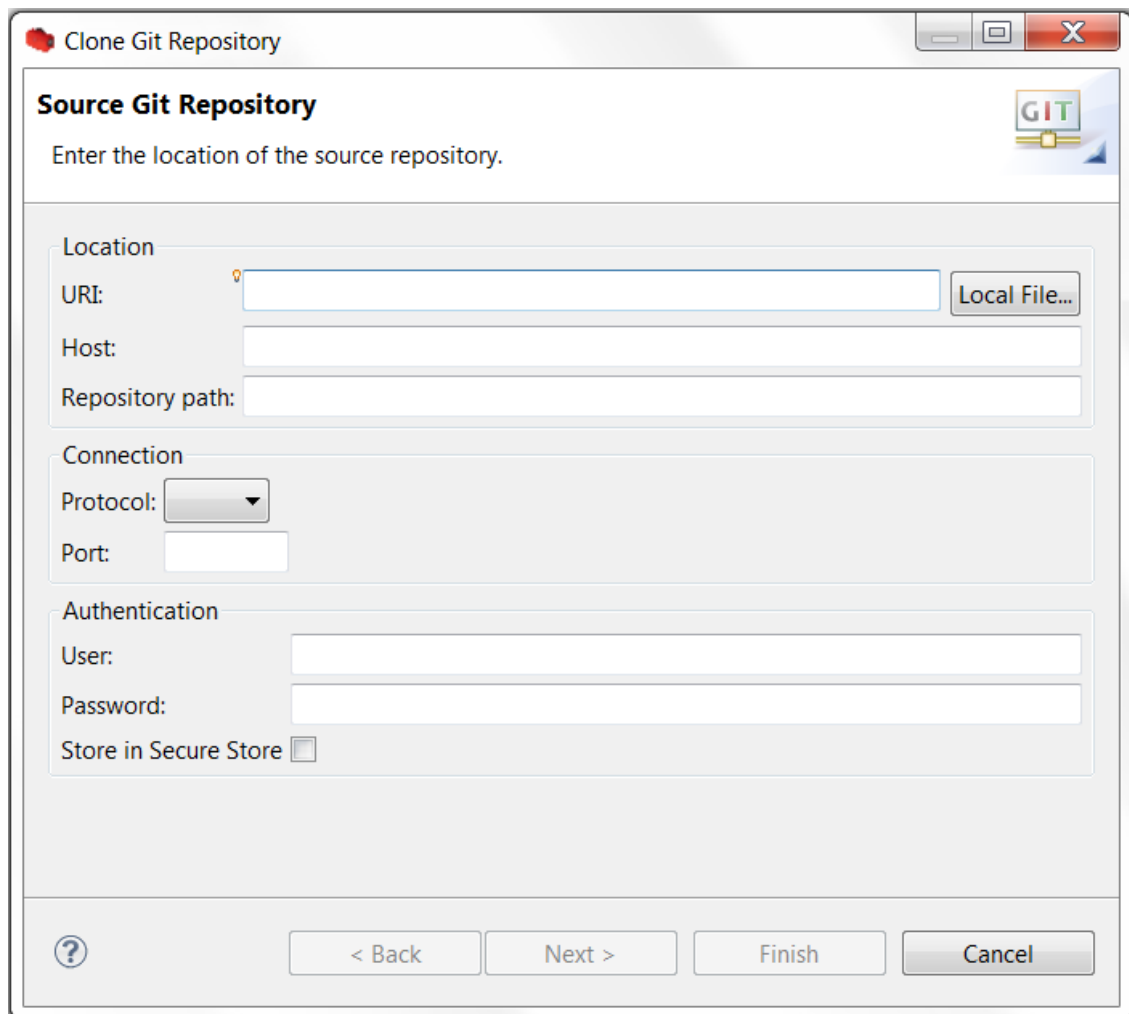
Ferramentas de linha de comando, como Git e SVN, oferecem uma abordagem direta para realizar operações de versionamento. Sistemas de integração contínua, como Jenkins e Travis CI, automatizam testes e integração contínua, muitas vezes, integrando-se a sistemas de controle de versão. Ferramentas de revisão de código, como *Crucible*, facilitam a revisão colaborativa de alterações no código-fonte, enquanto ferramentas de



rastreamento de problemas, como Jira e Redmine, ajudam a gerenciar problemas e melhorar a comunicação em torno de correções e aprimoramentos.

A integração do Eclipse com o Git permite que desenvolvedores gerenciem e controlem suas versões de código de forma eficiente diretamente do ambiente de desenvolvimento do Eclipse, como na Figura 12.

Figura 12 – O uso de ferramentas para o trabalho com GIT como o Eclipse



Fonte: <www.eclipse.org>. Acesso em: 15 abr. 2024.

Alguns outros exemplos de ferramentas de versionamento:

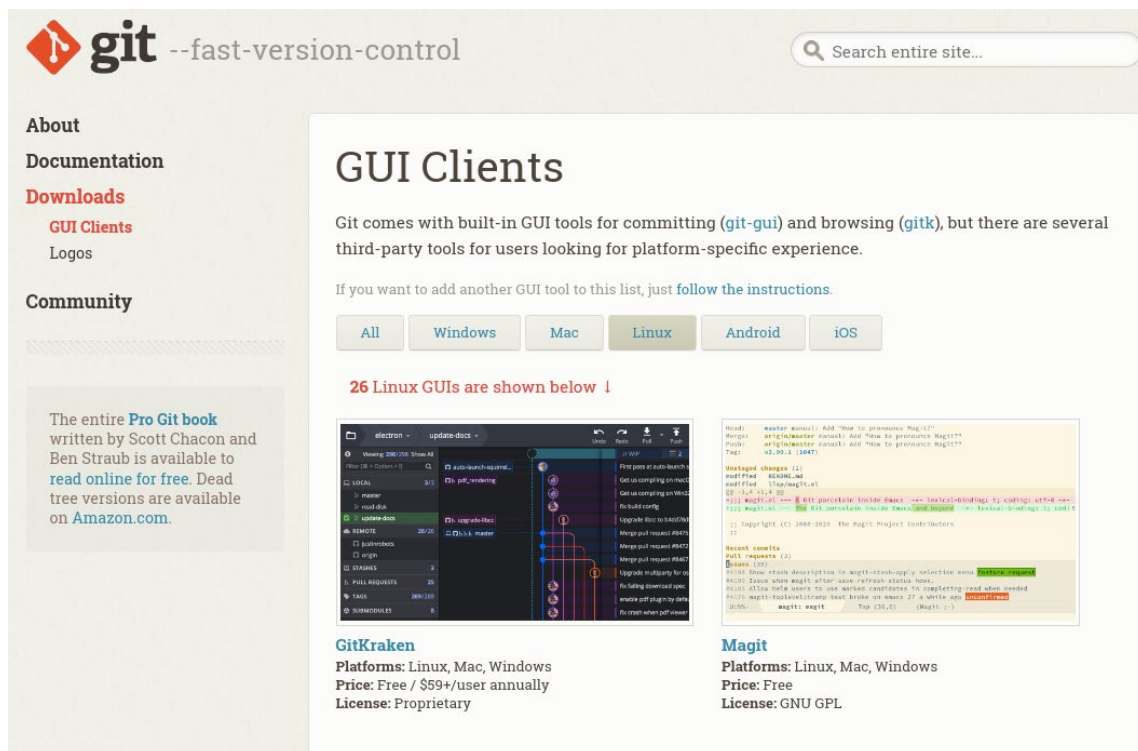
- **SourceTree**: uma interface gráfica de usuário (GUI) para o Git que facilita a visualização e interação com repositórios Git, especialmente útil para usuários que preferem uma abordagem visual.
- **GitKraken**: outra GUI para Git que oferece uma experiência visual para interagir com repositórios Git, incluindo recursos como integração com GitHub e GitLab.



- **TortoiseGit**: uma extensão do Windows Explorer que fornece integração com o Git, permitindo que os usuários realizem operações do Git diretamente no explorador de arquivos.

Conforme a imagem da Figura 13, o próprio sítio do GIT fornece uma ampla possibilidade de programas para trabalho.

Figura 13 – O uso de ferramentas para o trabalho com GIT



Fonte: <<https://git-scm.com/download/gui/linux>>. Acesso em: 15 abr. 2024.

TEMA 5 – GESTÃO DE *BACKUP* EM DEVOPS

Entender a estrutura de montar *backup* é essencial para garantir a segurança de seu processo, mas temos também de entender de infraestrutura como IAC.

5.1 O que é Infraestrutura como Código (IaC)?

Infraestrutura como Código (IaC) refere-se à gestão e provisionamento de recursos de infraestrutura por meio de código, substituindo processos manuais. Essa abordagem tem ganho crescente popularidade. Com o IaC, são criados arquivos de configuração contendo as especificações da infraestrutura,



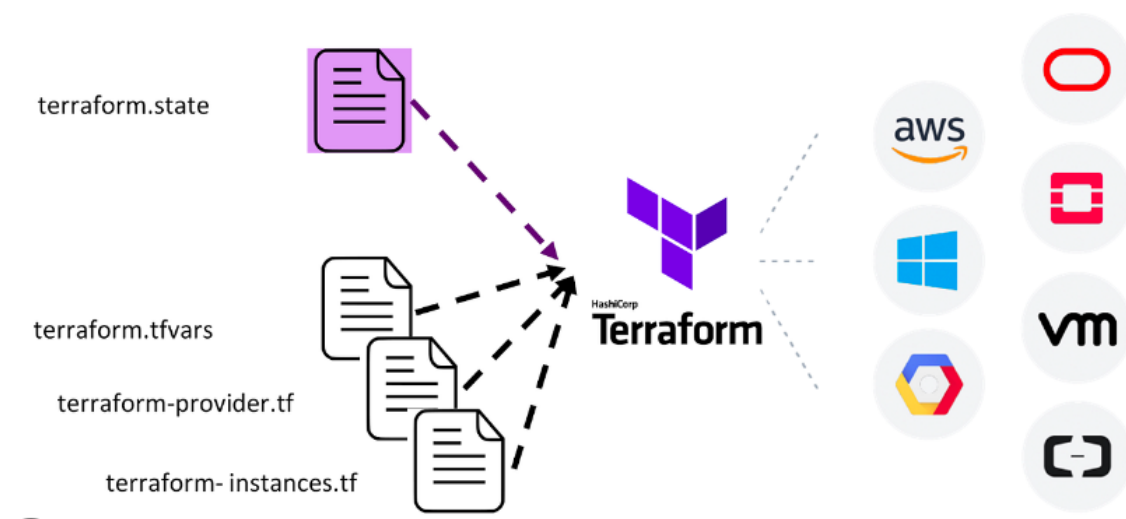
simplificando a edição e distribuição das configurações. Além disso, o IaC assegura a consistência na provisionamento do ambiente.

Ao codificar e documentar as especificações de configuração, o IaC facilita o gerenciamento de configuração e previne alterações não documentadas ou ad-hoc.

O controle de versão desempenha um papel crucial no IaC, e os arquivos de configuração devem ser tratados como qualquer outro arquivo de código-fonte, estando sujeitos ao controle de versionamento. A implementação de infraestrutura como código permite a divisão da infraestrutura em componentes modulares, que podem ser combinados de diversas maneiras por meio de automação.

Automatizar o provisionamento de infraestrutura com IaC significa que os desenvolvedores não necessitam gerenciar manualmente servidores, sistemas operacionais, armazenamento e outros componentes toda vez que desenvolvem ou implantam um aplicativo. A codificação da infraestrutura fornece um modelo para o provisionamento e, embora ainda possa ser feito manualmente, a automação torna esse processo mais eficiente e ágil.

Figura 14 – Esquema básico sobre o funcionamento de IAC, usando como exemplo *Terraform*



5.2 Automação do processo de *backup*

A automação do processo em *backup* é uma abordagem que envolve a implementação de ferramentas e *scripts* automatizados para realizar operações de *backup* de forma consistente e eficiente. Essa prática visa reduzir a



dependência de intervenção manual, minimizar erros humanos e assegurar que os *backups* sejam executados conforme políticas predefinidas, utilizando ferramentas de automação, os *backups* podem ser agendados em intervalos regulares, proporcionando uma execução previsível e alinhada com as necessidades da organização.

Essa automação oferece benefícios significativos, tais como a gestão centralizada de *backups* em ambientes complexos, a notificação automática sobre o *status* dos *backups*, a execução de *scripts* personalizados para tarefas específicas, e a integração com políticas de retenção de dados. Além disso, ela possibilita a geração automática de relatórios detalhados, fornecendo *insights* sobre o desempenho do processo de *backup*.

O DevOps emprega extensa automação, especialmente na gestão da infraestrutura, introduzindo o conceito de Infrastructure as Code (IaC), que, em uma tradução literal, significa Infraestrutura como Código. De acordo com a Hewlett Packard Enterprise (2019), a Infraestrutura como Código trata a infraestrutura como se fosse um *software* programável, permitindo que toda a infraestrutura necessária para executar um *software* seja criada por meio da execução de um código previamente desenvolvido.

Monteiro (2021) acrescenta que essa abordagem tecnológica viabiliza a reprodução de ambientes de forma idêntica, possibilitando que a equipe de desenvolvimento trabalhe em um ambiente de desenvolvimento e teste idêntico ao ambiente de produção. Isso resulta em uma significativa redução nos prazos de lançamento de *software* e, adicionalmente, diminui a ocorrência de erros decorrentes de testes em ambientes heterogêneos.

Essa prática é crucial para garantir a confiabilidade e eficácia das operações de *backup*, ao mesmo tempo em que reduz riscos associados a falhas humanas. A automação do processo em *backup* é uma parte integral da gestão moderna de dados, proporcionando consistência, escalabilidade e uma resposta rápida a alterações nas necessidades de proteção de dados.

A gestão de *backup* em DevOps é uma parte crucial da estratégia geral de garantir a agilidade e a colaboração inerentes à metodologia DevOps. Existem diversas ferramentas em DevOps que podem ser utilizadas para gestão de *backup*. Algumas delas incluem:

- **Veeam:** uma solução de *backup* e recuperação focada em ambientes virtuais. É conhecida por sua eficácia em ambientes VMware e Hyper-V.



- **Rubrik:** oferece uma plataforma de gestão de dados que inclui recursos avançados de *backup*, recuperação e orquestração.
- **Commvault:** uma solução abrangente de gestão de dados que inclui *backup*, recuperação, arquivamento e replicação.
- **Veritas NetBackup:** um sistema de *backup* corporativo que oferece recursos avançados de proteção de dados e recuperação.
- **Bacula:** uma solução de código aberto que fornece ferramentas de *backup*, recuperação e verificação de dados.
- **Duplicity:** uma ferramenta de *backup* baseada em Linux que realiza *backup* incremental e suporta várias opções de armazenamento.
- **Amanda Backup:** uma solução de *backup* de código aberto para ambientes Unix/Linux que oferece suporte a *backup* em fita, disco e nuvem.
- **BorgBackup:** a segurança, integridade e disponibilidade dos dados e sistemas. Ao integrar práticas de DevOps com gestão de *backup*, as equipes podem assegurar uma abordagem eficiente e automatizada para a proteção dos dados, enquanto uma ferramenta de *backup* de duplicada eficiente que oferece compressão e criptografia.
- **AWS Backup:** um serviço gerenciado pela *Amazon Web Services* (AWS) que simplifica a gestão de *backup* para os recursos da AWS.
- **Azure Backup:** um serviço de *backup* nativo da Microsoft Azure que oferece *backup* e recuperação para máquinas virtuais e serviços na nuvem.
- **GitLab:** embora seja conhecido principalmente como uma plataforma de controle de versão, o GitLab também oferece recursos de *backup* para repositórios Git.
- **Jenkins (para automação de backup):** embora seja uma ferramenta de integração contínua, o Jenkins pode ser configurado para automatizar tarefas de *backup* em um ambiente DevOps.

Existem vários tutoriais sobre *Backups* atualmente para Azure, AWS, o processo é mais operacional que acadêmico.



FINALIZANDO

Exploramos uma introdução ao controle de versão, abordando conceitos fundamentais. Em seguida, examinou o básico de Git e GitHub, destacando princípios de versionamento e colaboração. A terceira seção abordou técnicas avançadas, como *branching*, *merging* e *pull requests*. Também foram discutidas ferramentas de apoio ao versionamento. Por fim, o texto tratou da gestão de *backups* no contexto de DevOps.

Todo esse conteúdo é fundamental para entendimento do funcionamento de DevOps, não é todo o processo, mas é um passo a mais dado em sentido do objetivo.



REFERÊNCIAS

- AMAZON CODECOMMIT. 2023. Disponível em: <<https://aws.amazon.com/pt/codecommit/>>. Acesso em: 15 abr. 2024.
- AMAZON WEB SERVICES. **O que é DevOps?** Disponível em: <<https://aws.amazon.com/pt/devops/what-is-devops/>>. Acesso em: 15 abr. 2024.
- AZURE REPOS. 2023. Disponível em: <<https://azure.microsoft.com/pt-br/products/devops/repos>>. Acessado em: 15 abr. 2024.
- BORGES, E. N. **Conceitos e Benefícios do Test Driven Development**. Universidade Federal do Rio Grande do Sul. Disponível em: <<http://www.inf.ufrgs.br/~cesantin/TDD-Eduardo.pdf>>. Acesso em: 15 abr. 2024.
- CAELUM. **Test-Driven Development**. Disponível em: <<http://tdd.caelum.com.br/>>. Acesso em: 15 abr. 2024.
- CARMENA R. M. **How to teach Git**. Disponível em: <<https://rachelcarmena.github.io/2018/12/12/how-to-teach-git.html>>. Acesso em: 15 abr. 2024.
- CHACON, S.; STRAUB, B. **Pro Git**. 2. ed. Apress, 2014.
- CUKIER, D. **DDD: introdução a Domain Driven Design**. 2010. Disponível em: <<http://www.agileandart.com/2010/07/16/ddd-introducao-a-domain-driven-design/>>. Acesso em: 15 abr. 2024.
- DEVMEDIA. **TDD: fundamentos do desenvolvimento orientado a testes**. Disponível em: <<http://www.devmedia.com.br/tdd-fundamentos-do-desenvolvimento-orientado-a-testes/28151>>. Acesso em: 15 abr. 2024.
- DEVMEDIA. **Test Driven Development: TDD Simples e Prático**. 2019. Disponível em: <<https://www.devmedia.com.br/test-driven-development-tdd-simples-e-pratico/18533>>. Acesso em: 15 abr. 2024.
- ELEMARJR. **BDD na prática: parte 1 – Conceitos básicos e algum código**. 2012. Disponível em: <<https://elemarjr.wordpress.com/2012/04/11/bdd-na-prtica-parte-1-conceitos-bsicos-e-algum-cdigo/>>. Acesso em: 15 abr. 2024.
- ENAP. **Curso: O papel do DevOps na Transformação Digital dos Serviços Públicos**. 2020.



FREECODECAMP. Disponível em: <<https://www.freecodecamp.org/>>. Acessado em: 15 abr. 2024.

FREEMAN, S. PRYCE, N. **Desenvolvimento de Software Orientado a objetos, guiado por Testes**. Rio de Janeiro: Alta Books, 2012.

GASPARETO, O. **Test Driven Development**. Universidade Federal do Rio Grande do Sul. Disponível em: <<http://www.inf.ufrgs.br/~cesantin/TDD-Otavio.pdf>>. Acesso em: 15 abr. 2024.

GIT-FAST-VERSION-CONTROL. 2023. Disponível em: <<https://git-scm.com/>>. Acessado em: 15 abr. 2024.

GOMES, A. F. **Desenvolvimento Ágil com Kanban**. Disponível em: <http://www.devmedia.com.br/websys.5/webreader.aspat=6&artigo=2955&revisita=javamagazine_84#a-2955> Acesso em: 15 abr. 2024.

KIM, G. et al. **The DevOPS Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations**. [S. l.: s. n.], 2016.

LUZ, S. F. da. **Relação entre projetos ágeis e área de processos**. Curitiba: Intersaberes, 2021. (MBV)

MASCHIETTO, L. G. et al. **Desenvolvimento de software com metodologias ágeis**. Porto Alegre: SAGAH, 2020. (MBV)

MONTEIRO, E. R. et al. **DevOps**. Porto Alegre: SAGAH. 2021. (MBV)

MUNIZ, A. et al. **Jornada DevOps: unindo cultura ágil, Lean e tecnologia para entrega de software com qualidade**. Rio de Janeiro: Brasport, 2019. (BVP)

MUNIZ, A. et al. **Jornada Azure DevOps. Unindo teoria e prática com o objetivo de acelerar o aprendizado do Azure DevOps para quem está iniciando**. Rio de Janeiro: Brasport, 2021. (BVP)

NETFLIX. 2023. Disponível em: <<https://www.simform.com/blog/netflix-devops-case-study/>>. Acesso em: 20 dez. 2023.

PRESSMAN, R. S.; MAXIM, B. **Software Engineering: a practitioner's approach**. 8. ed. McGraw-Hill, 2014.

SBROCCO, J. H. T. de C. **Metodologias ágeis: engenharia de software sob medida**. 1. ed. São Paulo: Érica, 2012. (MBV)



SOMMERVILLE, I. **Engenharia de *software***. 10. Edição. São Paulo: Pearson, 2011. 544 p. (BVP)

W3C. Disponível em: <<https://www.w3schools.in/mvc-architecture>>. Acesso em: 15 abr. 2024.

WESLEY P. **Sistemas de controle de versão: SVN e Git**. 2017.